
sgcGrid

Getting Started

Build your first Delphi data grid, one small demo at a time

About this guide

This guide is for Delphi developers who know how to build a VCL form and want to show data in a fast, modern grid with sgcGrid. You do not need to know anything about sgcGrid yet. Every demo is built the usual Delphi way: you drop a component on a form, set its properties in the Object Inspector and write a few lines of Pascal.

The guide follows the five QuickStart demos. Each demo adds one idea to the previous one, from a grid with six rows to a small orders screen with groups, totals, search, filters and export. Chapter 4 walks through them one at a time: what you will build, the components on the form, the code, how it works and a few changes to try.

Read it with Delphi open next to it. Open a demo, run it, play with the grid, then read its section and try the suggested changes. Chapters 1 to 3 explain what sgcGrid is, how to install it and the ideas behind it, chapter 5 lists the main components, chapter 6 collects common problems and their fixes, and chapter 7 shows where to go next.

The demos are in the `Demos\Grid\Delphi\00.QuickStart` folder of your sgcUI installation. Each demo has its own subfolder, from `01.HelloGrid` to `05.OrdersApp`, with a Delphi project (`.dpr`) that you can open and run directly.

The screenshots were taken on a machine with Spanish regional settings, so they show numbers such as `1.499,00` and month names such as `ene.` and `feb.` . Numbers and dates in the grid follow the Windows regional settings of the machine running the demo, so on an English system the same cells read `1,499.00` and `Jan.`

Contents

1. What sgcGrid is

The idea in one minute
Your first grid in a dozen lines
What you need
sgcGrid and the rest of sgcUI
Coming from TStringGrid or TDBGrid

2. Install and the palette

Editions
Installing
Runtime and design packages
The SGC Grid palette page
Where the demos are
Running your first demo

3. Core concepts

Columns describe, Data holds
Every row has a key
Data types and display formats
Feeding the grid
Features are properties
Format rules and flashing
Themes
Threads
Editing is switched on, not off
Summary: who does what

4. The QuickStart demos

4.1 Demo 01: Hello Grid
4.2 Demo 02: Columns and Formatting
4.3 Demo 03: Editing and Events
4.4 Demo 04: DataSet
4.5 Demo 05: Orders App

5. Main components

Reference table
The grid controls
Data and sources
Alarm list and log viewer
Asking the grid in plain language
The sgcWebSockets bridge

6. Common problems

Compiling and editions
Rows and values
Threads
DataSets

7. Where next

The full demos
Help topics
On the website
Getting help

1. What sgcGrid is

sgcGrid is a data grid for Delphi. It shows rows and columns of data in a VCL or FireMonkey application, and it does the things users expect from a modern grid out of the box: sorting by a click on a header, typed columns with number and date formats, conditional formatting, grouping with totals, a search box, column filters, export to CSV and Excel, and live updates that flash the cells that change.

You use it like any other Delphi control. You drop a `TsgcVCLGrid` on a form, describe its columns in the Object Inspector and add rows from code, or connect it to a `TDataSet` with one non-visual component. Most features are switched on by setting a property, not by writing code.

This guide is for Delphi developers who are new to sgcGrid. You know Delphi well and you have probably used a `TStringGrid` or a `TDBGrid`. By the end you will know how the pieces fit together, which properties to reach for first, and how to avoid the mistakes that most people make in their first hour.

The idea in one minute

An sgcGrid grid has three parts, and once you see them the rest of the guide is easy to follow.

- **Columns describe the data.** The `Columns` collection says which fields the grid has, what kind of value each one holds (`DataType`) and how to show it (`DisplayFormat` , `Alignment` , `CellKind`). You usually design it at design time, in the Columns editor.
- **Data holds the rows.** Every grid owns a `TsgcGridData` object, reached through `Grid.Data` . It stores one value per column for each row, keeps the values with their real type (a number stays a number, a date stays a date) and knows each row by a text key.
- **The control paints and reacts.** `TsgcVCLGrid` paints the data, only the rows that are visible, and adds the mouse, the keyboard, the scroll bars and the built in features: sorting, the group panel, the find panel, filters, the footer with summaries.

Your code talks to `Grid.Data` to add, change and remove rows. The grid repaints by itself. You never paint cells and you never keep a separate list of strings in step with the screen.

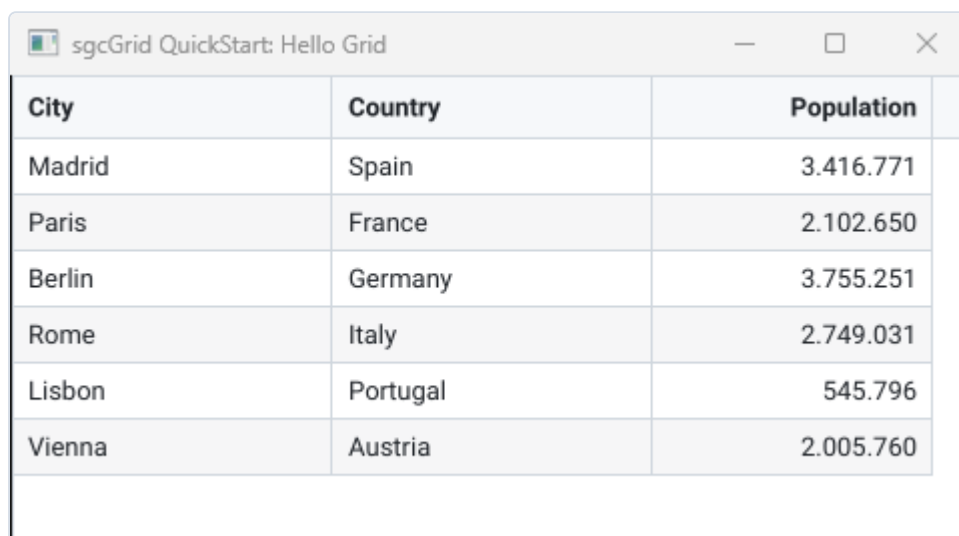
Your first grid in a dozen lines

This is the complete code of the first QuickStart demo. The three columns (City, Country and Population) are designed in the Object Inspector, so the only code adds six rows:

```

procedure TfrmHelloGrid.FormCreate(Sender: TObject);
begin
  { BeginUpdate / EndUpdate repaint the grid once, after the last row,
    instead of once per row. Always use them when adding many rows. }
  Grid.Data.BeginUpdate;
  try
    { AppendRow(Key, [City, Country, Population]) }
    Grid.Data.AppendRow('Madrid', ['Madrid', 'Spain', 3416771]);
    Grid.Data.AppendRow('Paris', ['Paris', 'France', 2102650]);
    Grid.Data.AppendRow('Berlin', ['Berlin', 'Germany', 3755251]);
    Grid.Data.AppendRow('Rome', ['Rome', 'Italy', 2749031]);
    Grid.Data.AppendRow('Lisbon', ['Lisbon', 'Portugal', 545796]);
    Grid.Data.AppendRow('Vienna', ['Vienna', 'Austria', 2005760]);
  finally
    Grid.Data.EndUpdate;
  end;
end;

```



City	Country	Population
Madrid	Spain	3.416.771
Paris	France	2.102.650
Berlin	Germany	3.755.251
Rome	Italy	2.749.031
Lisbon	Portugal	545.796
Vienna	Austria	2.005.760

Demo 01 running. A click on a header sorts the column; the population column is right aligned and formatted with thousands separators by its `DisplayFormat`.

Each `AppendRow` call takes a key, here the city name, and then one value per column in the order of the `Columns` collection. Run the demo and click the Population header: the rows are sorted as numbers, not as text, because the column says its `DataType` is `gdtInteger`. You wrote no code for that.

What you need

- **Delphi.** sgcGrid ships packages for Delphi 7, 2007, 2009, 2010, XE to XE8, 10 to 10.4, 11, 12 and 13. The QuickStart demos are VCL applications and compile with all of them.
- **VCL or FireMonkey.** `TsgcVCLGrid` is the VCL grid, for Windows 32 and 64 bit. `TsgcFMXGrid` is the FireMonkey grid, from Delphi XE6, for Windows, macOS, iOS and Android, with the same properties, methods and events. This guide uses the VCL grid; everything you learn carries over to the FireMonkey one.
- **The sgcUI installation.** sgcGrid is part of the sgcUI suite. The installer puts the sgcGrid packages, the sgcUICore packages they depend on and the sgcEditors packages the grids require on your machine, and compiles them for the Delphi versions you choose. Chapter 2 explains the details.
- **For demo 04 only:** `TClientDataSet` and `MidasLib`, which ship with Delphi. Nothing else is needed: no database server, no DLL and no internet connection.

sgcGrid and the rest of sgcUI

sgcUI is a family of visual products for Delphi that share one foundation, sgcUICore. sgcUICore provides the vector canvas the grid paints on, the control base that schedules repaints, the shared colour type `TsgcUIColor` and the theme tokens. That is why the grid has a `Theme` property with the same values as the other sgcUI controls (`utDefault`, `utLight`, `utDark`, `utCustom`, `utHighContrast`), and why a grid placed next to an sgcUI chart or dashboard looks like it belongs there.

You do not have to use any other sgcUI product to use sgcGrid. The installer adds sgcUICore automatically whenever sgcGrid is selected, and it has no checkbox of its own.

Note. Colours in sgcGrid, such as the `ForeColor` of a format rule or the `ProgressColor` of a column, are `TsgcUIColor` values: a Cardinal holding alpha, red, green and blue, written `$AARRGGBB`. Opaque red is `$FFDC2626`. They are not `TColor` values, so `clRed` does not mean red here.

Coming from TStringGrid or TDBGrid

If you have used the grids that come with Delphi, a few differences are worth knowing before you start.

You are used to	In sgcGrid
<code>TStringGrid.Cells[Col, Row] := '...'</code>	Rows are added with <code>Grid.Data.AppendRow</code> and changed with <code>Grid.Data.UpdateRow</code> , by key. Values keep their type, so numbers sort as numbers and dates as dates.
Painting cells in <code>OnDrawCell</code>	Columns have a <code>DisplayFormat</code> , a <code>CellKind</code> (text, progress bar and more) and the grid has <code>FormatRules</code> for conditional formatting. Most screens need no paint code at all.
Writing your own sort code	A click on a header sorts. Ctrl and a click adds a second sort column.
<code>TDBGrid</code> with a <code>TDataSource</code>	A <code>TsgcGridSource_DataSet</code> connects the dataset to the grid. It loads the records into the grid, which then sorts, filters and groups in memory, and it follows the changes of the dataset.
Columns of a <code>TDBGrid</code>	Right click a <code>TsgcVCLGrid</code> in the designer and choose Import columns from DB grid... to copy the columns of an existing DB grid.

The full demo `MigrateFromDBGrid`, in the folder next to the QuickStart, maps the `TDBGrid` concepts one by one and then shows the features people usually look for in larger commercial grids, such as grouping, summaries, the filter row and the find panel.

2. Install and the palette

This chapter gets sgcGrid into your IDE and runs the first demo. If the **SGC Grid** page is already on your Tool Palette, skip to [Where the demos are](#).

Editions

sgcGrid is built from the same source for every edition. The edition only changes which features are compiled.

- **Registered** includes everything.
- **Trial** has the same features as Registered.
- **Community** is the free edition. It includes sgcGrid with sgcUICore and the sgcEditors runtime packages the grid needs. It leaves out in-place editing and the other features that change data (clipboard paste, undo and redo, inserting and deleting rows, the sgcEditors cell editors, writing back to a DataSet), tree rows, master detail rows, Excel export and the sgcWebSockets bridge. Sorting, typed columns, format rules, grouping with summaries, the find panel, column filters, the filter row, CSV and JSON, DataSet loading and themes are all included.

Watch out. Demos 01, 02 and 05 work in every edition. Demos 03 (editing events) and 04 (`SyncCursor` and `WriteBack` of the DataSet source) do not compile with the Community edition, because those events and properties are not part of it. With Community, read their sections in chapter 4 to learn the ideas, and run them with a Trial or Registered installation.

The help topic *Editions* (in the Grid section of the sgcUI help) has the full feature table.

Installing

One installer covers the whole sgcUI suite, and it installs only the products your licence includes.

1. Run the sgcUI installer and sign in with the user name and password of your eSeGeCe account.
2. Choose the Delphi versions to install into.
3. On the components page, select **sgcGrid**. The installer also installs sgcUICore, and the sgcEditors packages that sgcGrid requires.
4. If sgcWebSockets is installed on the machine, the page also offers **sgcGrid for sgcWebSockets** under the bridges. It adds the live feed sources described in chapter 5. You do not need it for this guide.
5. The installer copies the files, compiles the packages for each selected Delphi version in dependency order and installs the design time packages. It also adds the sgcUI `Source` folder to the library path of each selected IDE.

Everything goes into one folder, by default `Documents\esegece\sgcUI` (`sgcUI_Trial` for the trial and `sgcUI_Community` for the Community edition). The `Source`, `Packages` and `Demos` subfolders are shared by every sgcUI product.

Runtime and design packages

The installer builds the packages for you, but it helps to know what they are when you deploy with runtime packages or build them by hand. Each name ends with the Delphi version, for example `D13` for Delphi 13 and `D7` for Delphi 7.

Package	Kind	Contents
<code>sgcGridD13</code>	Runtime, framework neutral	The data, the columns, formatting, rendering, grouping, summaries, import and export, <code>TsgcGridSource_DataSet</code> , the alarm and log models.
<code>sgcGridVCLD13</code>	Runtime, VCL	<code>TsgcVCLGrid</code> and <code>TsgcVCLGridAlarmMenu</code> .
<code>sgcGridFMXD13</code>	Runtime, FireMonkey (XE6 and later)	<code>TsgcFMXGrid</code> and <code>TsgcFMXGridAlarmMenu</code> .
<code>dc1sgcGridD13</code> , <code>dc1sgcGridFMXD13</code>	Design time	Register the components on the SGC Grid palette page and add the component editors.
<code>sgcGridWSD13</code> , <code>dc1sgcGridWSD13</code>	Runtime and design time	The <code>sgcWebSockets</code> bridge: live feed sources and the AI query. Needs <code>sgcWebSockets</code> .

To build them yourself, open them from `sgcUI\Packages` and build in dependency order: `sgcUICore` first, then the `sgcEditors` runtime packages, then `sgcGrid` and `sgcGridVCL`, then install `dc1sgcGrid`. Add `sgcUI\Source` to the library path of your Delphi version.

The SGC Grid palette page

After the installation the Tool Palette has a page called **SGC Grid**. On a VCL form it shows these components:

Component	What it is for
<code>TsgcVCLGrid</code>	The grid control. The only visual component you need for this guide.
<code>TsgcGridSource_DataSet</code>	Fills a grid from any <code>TDataSet</code> and keeps it in step. Used in demo 04.
<code>TsgcGridAI</code>	Turns a question in plain language into filters, sorts and highlights.
<code>TsgcGridAlarmModel</code> , <code>TsgcGridAlarmLink</code> , <code>TsgcVCLGridAlarmMenu</code>	An ISA-18.2 alarm list shown in a grid, with its operator menu.
<code>TsgcGridLogModel</code> , <code>TsgcGridLogLink</code>	A log kept in a ring buffer and shown in a grid.
<code>TsgcGridDataStore</code>	Holds rows without showing them, as the source of detail bands. Not in Community.
<code>TsgcGridServerProvider_DataSet</code> , <code>TsgcGridServerProvider_Callback</code>	Server mode: the grid reads one page at a time and the database sorts, filters and groups. Not in Community.

On a FireMonkey form the same page shows `TsgcFMXGrid` and `TsgcFMXGridAlarmMenu` instead of the VCL controls. Chapter 5 describes each component in a few lines. With the bridge installed, the page also holds the feed sources of `sgcGridWS`.

Tip. Right click a `TsgcVCLGrid` on a form to see its component editor verbs. **Columns editor...** opens the Columns collection directly, the same editor you get by double clicking `Columns` in the Object Inspector.

Where the demos are

The `sgcGrid` demos are in `Demos\Grid` of your `sgcUI` installation. The Delphi demos are in `Demos\Grid\Delphi`: the `QuickStart` is the folder `00.QuickStart`, and every other folder is a full demo of one feature (chapter 7 lists them).

Folder	Project	What it shows
01.HelloGrid	HelloGrid.dpr	The smallest grid application: columns at design time, six rows from code
02.Columns	GridColumns.dpr	Typed columns, display formats, a check box column, a progress column and a format rule
03.Editing	GridEditing.dpr	In-place editing with pick lists, validation and a change log. Not with Community
04.DataSet	GridDataSet.dpr	A grid bound to a <code>TClientDataSet</code> that follows the navigator and writes edits back. Not with Community
05.OrdersApp	GridOrders.dpr	A small realistic screen: groups, totals, search, filters, status icons and CSV export

Every demo folder has the project, one form unit (`u<Name>.pas` and `.dfm`) and a `readme.html`. Every setting that can live in the form is in the `.dfm`, and each unit starts with a comment for beginners that explains the components and properties used. The `readme.html` in `00.QuickStart` is an index of the five demos.

Running your first demo

1. In Delphi, choose **File > Open Project** and open `Demos\Grid\Delphi\00.QuickStart\01.HelloGrid\HelloGrid.dpr`.
2. Open the form `uHelloGrid`. Click the grid and look at the Object Inspector: `Align` is `alClient`, and the `Columns` property holds three columns. Double click `Columns` to open the editor and select each column to see its `FieldName`, `Caption` and `DataType`.
3. Press **F9**. A small window with six cities opens.
4. Click the **Population** header to sort, click it again to reverse the order, and a third time to go back to the original order. Drag the edge of a header to resize a column.

Note. If the compiler stops with *File not found: sgcGrid_VCL.dcu*, the `sgcUI Source` folder (or the folder with the compiled units) is not in the library path of the Delphi version and platform you are building for. The installer adds it for every IDE it installs into; for a manual installation add it under **Tools > Options > Language > Delphi Options > Library**.

3. Core concepts

This chapter explains the ideas behind `sgcGrid`. You can read it before the demos or come back to it after you have run them. Every idea here shows up in at least one QuickStart demo, and the code snippets that are not taken from a demo were compiled in a small test program.

Columns describe, Data holds

A grid has two collections that you work with all the time, and they have different jobs.

- **Grid.Columns** is a collection of `TsgcGridColumn` items. Each column has a `FieldName`, the name you use in code, and a `Caption`, the text of its header. The other properties of a column say what kind of value it holds and how to show it. You normally fill `Columns` at design time, with the Columns editor, and the grid streams them into the `.dfm`.
- **Grid.Data** is the `TsgcGridData` that holds the rows. Each row stores one `Variant` per column, in the order of the `Columns` collection. Data also sorts, filters and counts, and it is the object you call from code.

Columns can also be created in code. `Columns.Add` takes the field name, the caption and optionally the data type, and returns the new column so you can set more properties on it:

```
procedure TfrmSnippets.CreateGrid;
var
  oGrid: TsgcVCLGrid;
begin
  oGrid := TsgcVCLGrid.Create(Self);
  oGrid.Parent := Self;
  oGrid.Align := alClient;
  { the columns: FieldName, Caption and DataType }
  oGrid.Columns.Add('symbol', 'Symbol');
  oGrid.Columns.Add('price', 'Price', gdtFloat).DisplayFormat := '0.00';
  oGrid.Columns.Add('volume', 'Volume', gdtInteger);
  { one row per key, the values in column order }
  oGrid.Data.AppendRow('MSFT', ['MSFT', 412.5, 1200]);
  oGrid.Data.AppendRow('AAPL', ['AAPL', 228.1, 3400]);
  FGrid := oGrid;
end;
```

The units this needs are `sgcGrid_Columns` (for `gdtFloat` and the other data types), `sgcGrid_Data` and `sgcGrid_VCL`. When you drop a grid on a form the IDE adds `sgcGrid_VCL` for you; add the others to the `uses` clause when your code needs their types.

Every row has a key

The first parameter of `AppendRow` is the **key** of the row: a text that names it, such as a customer code, a city or an order number. The key is how you find the row again. `UpdateRow`, `UpdateCell` and `DeleteRow` all take the key, and so does the `DataSet` source when it updates rows in place.

- Keys must be unique, and they are case-sensitive: `'MSFT'` and `'msft'` are two different rows.
- The key is not a column. If you want to see it, also pass it as the value of a column, as every QuickStart demo does.

- `Grid.Data.KeyOf(aRow)` gives you the key of a row on the screen, which is what demo 03 does to say which task was edited.

Tip. Rows are stored in the order you add them. When the user sorts or filters, the grid shows a *view* of the stored rows. Event parameters such as `aRow` are view rows, the row as the user sees it. Use the key, not the row number, whenever you need to find the same row later.

Data types and display formats

Three column properties decide what the user sees, and it is worth keeping them apart.

Property	What it decides	Values
<code>DataType</code>	What kind of value the column holds. It decides how the column sorts (numbers as numbers, dates as dates), how it is aligned by default and which summaries it supports.	<code>gdtText</code> , <code>gdtInteger</code> , <code>gdtFloat</code> , <code>gdtCurrency</code> , <code>gdtDateTime</code> , <code>gdtBoolean</code>
<code>DisplayFormat</code>	How the value is turned into text, with the Delphi <code>FormatFloat</code> and <code>FormatDateTime</code> patterns. The value itself does not change.	'#,##0.00', 'dd mmm yyyy', '0" %"'
<code>CellKind</code>	What the cell paints.	<code>gcdText</code> (the default), <code>gcdProgress</code> (a bar) and others

A `gdtBoolean` column with an empty `DisplayFormat` paints a check box. Give it a `DisplayFormat` such as 'Yes;No' and it shows text instead.

Formatted numbers and dates follow the regional settings of the machine, through `Data.FormatSettings`. That is why the screenshots in this guide show a decimal comma. If you need the same output everywhere, call `Grid.Data.UseInvariantFormat`: it sets a dot as the decimal separator, a comma for thousands, `yyyy-mm-dd` for dates and `hh:nn:ss` for times.

Feeding the grid

There are three ways to put rows into a grid, and you can pick the one that suits your data.

AppendRow: you have the values

`AppendRow(Key, [values])` adds a row with one value per column, in column order. Wrap many calls in `BeginUpdate` and `EndUpdate` so the grid repaints once at the end. Demos 01, 02, 03 and 05 fill their grids this way.

UpdateRow: the values change

`UpdateRow` takes field names instead of positions, so a message that only carries a new price only changes the price. `UpdateCell` changes one value, and `DeleteRow` removes a row:

```

procedure TfrmSnippets.UpdatePrices;
begin
  { by key and field name: only the fields you name change }
  FGrid.Data.UpdateRow('MSFT', ['price', 'volume'], [413.1, 1350]);
  FGrid.Data.UpdateCell('AAPL', 'price', 229.4);
  { a key that does not exist yet is inserted (Data.AutoInsert) }
  FGrid.Data.UpdateRow('NVDA', ['symbol', 'price'], ['NVDA', 131.2]);
  FGrid.Data.DeleteRow('AAPL');
end;

```

When `UpdateRow` gets a key that does not exist yet, it inserts a new row, unless you set `Grid.Data.AutoInsert` to `False`; then the update is ignored. Field names that do not match a column are ignored too.

A source component: the data lives somewhere else

A `TsgcGridSource_DataSet` fills the grid from any `TDataSet`: a `TClientDataSet`, a FireDAC query or anything else. It loads every record into the grid. With a `KeyField`, a reload updates changed rows in place, appends new keys and deletes missing ones, and with `LiveUpdates` (on by default) the grid follows every posted change of the dataset. When the grid has no columns, `AutoColumns` (also on by default) creates one per field. Demo 04 does all of this at design time; in code it is five lines:

```

procedure TfrmSnippets.BindDataSet(aDataSet: TDataSet);
begin
  FSource := TsgcGridSource_DataSet.Create(Self);
  FSource.DataSet := aDataSet;
  FSource.Grid := FGrid;
  FSource.KeyField := 'ID';
  FSource.Active := True;
end;

```

Besides the `DataSet` source, the unit `sgcGrid_IO` loads and saves JSON and CSV (`GridLoadFromCSVFile`, `GridSaveToJSONFile` and their siblings), and the `sgcWebSockets` bridge has sources that write the rows of a `WebSocket`, `MQTT` or `SSE` feed into the grid.

Features are properties

Most of what makes `sgcGrid` feel like a finished product is already inside the control. You switch a feature on by setting a property, usually at design time:

Property	What the user gets
<code>Columns[i].SortOrder</code>	The column starts sorted. Users sort by clicking headers anyway.
<code>FrozenColumns</code>	The first columns stay in place when scrolling to the right.
<code>RowHeader</code>	A strip at the left of the rows: row numbers, an indicator or check boxes.
<code>ShowFindPanel</code> , <code>FindOnlyMatches</code>	A search box above the grid that highlights matches, or shows only the matching rows. Ctrl+F focuses it.
<code>ShowFilterButtons</code> , <code>ShowFilterRow</code>	A filter button in every header, and a row under the header to type a filter per column.
<code>ShowGroupPanel</code> , <code>Columns[i].GroupIndex</code>	The group bar: drag a header onto it to group by that column. <code>GroupIndex</code> groups at design time.
<code>ShowFooter</code> , <code>Columns[i].SummaryType</code>	A footer with a sum, a count or another summary per column, repeated on each group row.
<code>SelectionMode</code>	Row selection (the default), single cell, several rows or a cell range.

Demo 05 turns most of them on and has only a few lines of code, none of which is about these features.

Format rules and flashing

`Grid.FormatRules` is a collection of conditional formats. Each rule names a field, an operator (`gfoEqual`, `gfoLess`, `gfoGreater` and others) and a value, and says what to do when a cell matches: `ForeColor`, `BackColor`, `Bold`, an `Icon` with its `IconColor`, and `ApplyToRow` to paint the whole row. Demo 02 marks low stock in red, and demo 05 shows a coloured circle for each order status. Rules can also be added in code:

```
procedure TfrmSnippets.SetAppearance;
begin
    FGrid.Theme := utDark;
    FGrid.RowHeader := [grhNumbers];
    { FieldName, Operator, Value, BackColor, ForeColor (ARGB) }
    FGrid.FormatRules.Add('price', gfoGreater, '400', 0, $FF16A34A);
    { '.' as decimal separator and yyyy-mm-dd, whatever the locale }
    FGrid.Data.UseInvariantFormat;
end;
```

A `BackColor` of 0 means "leave the background alone". The types come from `sgcUICore_Theme` (`utDark`), `sgcGrid_Render` (`grhNumbers`) and `sgcGrid_Format` (`gfoGreater`).

The grid also flashes cells whose value changes, which matters as soon as data is updated live. `FlashChanges` is `True` by default: a value that went up is tinted with `FlashUpColor`, one that went down with `FlashDownColor`, any other change with `FlashChangedColor`, and the tint fades out over `FlashDuration` milliseconds (800 by default).

Themes

The grid uses the theme tokens of `sgcUICore`. `Theme` is `utDefault` by default, which follows the application theme; set it to `utLight` or `utDark` to force one. `AccentColor` sets the colour of the selection and of the change flash. For the density, use `FontSize`, `RowHeight` (28 by default), `HeaderHeight` and `AlternateRows`. Try `Theme = utDark` on demo 01: it is a one property change.

Threads

This is the part that surprises most people coming from other grids. `TsgcGridData` is thread-safe: every public reader and writer takes an internal lock, so a worker thread can call `AppendRow`, `UpdateRow`, `UpdateCell` and `DeleteRow` directly, without `Synchronize`. The grid does not paint on that thread. It marks itself dirty and repaints in the main thread, at most `MaxFPS` times per second (60 by default), so thousands of updates per second do not flood the user interface.

```
type
  TPriceThread = class(TThread)
  private
    FData: TsgcGridData;
  protected
    procedure Execute; override;
  public
    constructor Create(aData: TsgcGridData);
  end;

constructor TPriceThread.Create(aData: TsgcGridData);
begin
  FData := aData;
  inherited Create(False);
end;

procedure TPriceThread.Execute;
begin
  while not Terminated do
  begin
    { UpdateRow is thread-safe: no Synchronize, no TThread.Queue }
    FData.UpdateRow('MSFT', ['price'], [400 + Random(2000) / 100]);
    Sleep(100);
  end;
end;
```

Start it with `TPriceThread.Create(Grid.Data)`. Stop it before the form, and with it the grid and its data, is freed:

```
destructor TfrmSnippets.Destroy;
begin
  if Assigned(FThread) then
  begin
    { stop the thread before the grid and its data are freed }
    FThread.Terminate;
    FThread.WaitFor;
    FThread.Free;
  end;
  inherited;
end;
```

Three rules go with this freedom:

- Change **columns** from the main thread. If you must change them while writer threads run, wrap the change in `Grid.Data.Lock` and `Grid.Data.Unlock`.
- The data raises its `OnChange` on the thread that wrote, after the lock is released. Do not touch controls there; use `TThread.Queue` to get to the main thread.
- Events that run with the data lock held, such as `OnFilterRow`, must never wait for the main thread and never touch a control.

The QuickStart demos fill the grid from the main thread, which is always fine. The full demo `LiveTicker` shows a worker thread writing about a thousand updates per second.

Editing is switched on, not off

A new grid is read-only: `ReadOnly` is `True` by default. Set it to `False` and the user can edit cells in place with F2, Enter, a double click or by simply typing. Each column has `Editable` (to protect a column such as a code) and an optional `PickList` that turns the editor into a combo box. Three events let your code take part:

- `OnValidate` receives the new value, already converted to the column type, and can refuse it with `aAccept := False`.
- `OnEditError` fires when the typed text cannot be converted, for example letters in a number column.
- `OnEdited` fires after a change, with the old and the new value.

Demo 03 uses all three. In-place editing is not part of the Community edition.

Summary: who does what

Piece	Its job	You touch it
<code>TsgcVCLGrid</code>	Paints, scrolls, sorts on a click, shows the find panel, filters, groups and footer	At design time, through properties
<code>Grid.Columns</code>	Field names, captions, types, formats, cell kinds, summaries	In the Columns editor
<code>Grid.FormatRules</code>	Conditional colours, bold text and icons	In its editor, or with <code>FormatRules.Add</code>
<code>Grid.Data</code>	Stores the rows by key, sorts, filters, counts, thread-safe	From code: <code>AppendRow</code> , <code>UpdateRow</code> , <code>DeleteRow</code>
<code>TsgcGridSource_DataSet</code>	Copies a <code>TDataSet</code> into the grid and keeps it in step	At design time: <code>DataSet</code> , <code>Grid</code> , <code>KeyField</code> , <code>Active</code>
Grid events	Tell you what the user did: edits, view changes, clicks	Event handlers in your form

4. The QuickStart demos

The five demos build on each other. Open them in order, run each one, then read its section with the form open in the designer. Every setting that can be made in the Object Inspector is made there, so the code in each demo is short.

Demo	Project	Idea	Editions
01.HelloGrid	HelloGrid.dpr	Columns at design time, rows from code	All
02.Columns	GridColumns.dpr	Data types, display formats, cell kinds, a format rule	All
03.Editing	GridEditing.dpr	In-place editing, validation, change log	Not Community
04.DataSet	GridDataSet.dpr	A TDataSet shown and edited through the grid	Not Community
05.OrdersApp	GridOrders.dpr	Groups, totals, search, filters, icons, CSV export	All

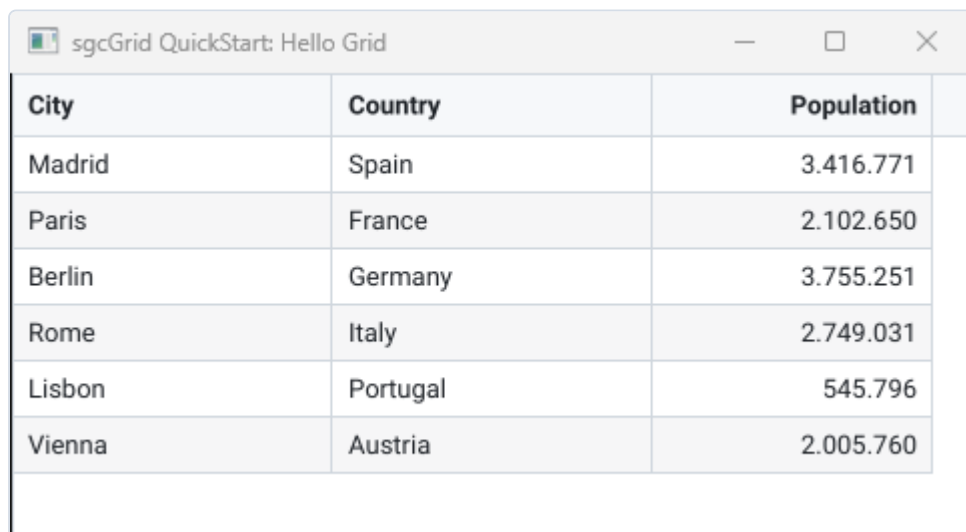
Tip. To see every property of a form at once, right click the form in the designer and choose **View as Text**. The DFM excerpts in this chapter come from that view, sometimes shortened with a `// ...` line. Choose **View as Form** to go back to the designer.

4.1 Demo 01: Hello Grid

Folder `01.HelloGrid`, project `HelloGrid.dpr`, form `TfrmHelloGrid` in `uHelloGrid.pas`. Works in every edition.

What you will build

The smallest grid application there is: one grid that fills the window, three columns and six European cities with their country and population. The columns are designed in the Object Inspector and the rows are added in `FormCreate`. Sorting, resizing columns and keyboard navigation come with the control.



City	Country	Population
Madrid	Spain	3.416.771
Paris	France	2.102.650
Berlin	Germany	3.755.251
Rome	Italy	2.749.031
Lisbon	Portugal	545.796
Vienna	Austria	2.005.760

Figure 4.1. Hello Grid. The population is right aligned and has thousands separators because the column is a `gdtInteger` column with the `DisplayFormat #,##0`.

Components on the form

Component	Class	Properties you set
Grid	TsgcVCLGrid	Align = alClient. Columns: city (Caption City, Width 160), country (Country, 160), population (Population, DataType = gdtInteger, Width 140, DisplayFormat = '#,##0')

One component, and its `Columns` collection is the interesting part. This is the grid in `uHelloGrid.dfm`:

```
object Grid: TsgcVCLGrid
  Left = 0
  Top = 0
  Width = 480
  Height = 232
  Align = alClient
  TabOrder = 0
  Columns = <
    item
      FieldName = 'city'
      Caption = 'City'
      Width = 160
    end
    item
      FieldName = 'country'
      Caption = 'Country'
      Width = 160
    end
    item
      FieldName = 'population'
      Caption = 'Population'
      DataType = gdtInteger
      Width = 140
      DisplayFormat = '#,##0'
    end>
  FormatRules = <>
end
```

To build it yourself: drop a `TsgcVCLGrid` from the SGC Grid page, set `Align` to `alClient`, double click `Columns` in the Object Inspector and add three items. Select each item to set its `FieldName`, `Caption` and, for the population, `DataType` and `DisplayFormat`. The columns appear in the designer as you add them.

The code

The `uses` clause names the two `sgcGrid` units the form needs: `sgcGrid_VCL` for the control, added by the IDE when you drop the grid, and `sgcGrid_Data` for the data object behind `Grid.Data`.

```
uses
  Classes, Controls, Forms,
  // sgc
  sgcGrid_Data, sgcGrid_VCL;
```

And this is every line of code in the demo:

```

procedure TfrmHelloGrid.FormCreate(Sender: TObject);
begin
  { BeginUpdate / EndUpdate repaint the grid once, after the last row,
    instead of once per row. Always use them when adding many rows. }
  Grid.Data.BeginUpdate;
  try
    { AppendRow(Key, [City, Country, Population]) }
    Grid.Data.AppendRow('Madrid', ['Madrid', 'Spain', 3416771]);
    Grid.Data.AppendRow('Paris', ['Paris', 'France', 2102650]);
    Grid.Data.AppendRow('Berlin', ['Berlin', 'Germany', 3755251]);
    Grid.Data.AppendRow('Rome', ['Rome', 'Italy', 2749031]);
    Grid.Data.AppendRow('Lisbon', ['Lisbon', 'Portugal', 545796]);
    Grid.Data.AppendRow('Vienna', ['Vienna', 'Austria', 2005760]);
  finally
    Grid.Data.EndUpdate;
  end;
end;

```

How it works

1. The form loads from its DFM, and the grid creates its three columns from the `Columns` collection.
2. `FormCreate` calls `Grid.Data.BeginUpdate`, so the six rows that follow cause one repaint instead of six. The `try / finally` makes sure `EndUpdate` runs even if something fails.
3. Each `AppendRow` stores a row under its key (the city) with three values, in the order of the columns: city, country, population. The population is passed as an integer and stays an integer.
4. `EndUpdate` repaints the grid. The population column formats each number with `#,##0`, using the thousands separator of your regional settings.
5. You click a header. The grid sorts by that column, comparing numbers as numbers because of `DataType`. A second click reverses the order, a third one goes back to the order in which the rows were added.

Try this

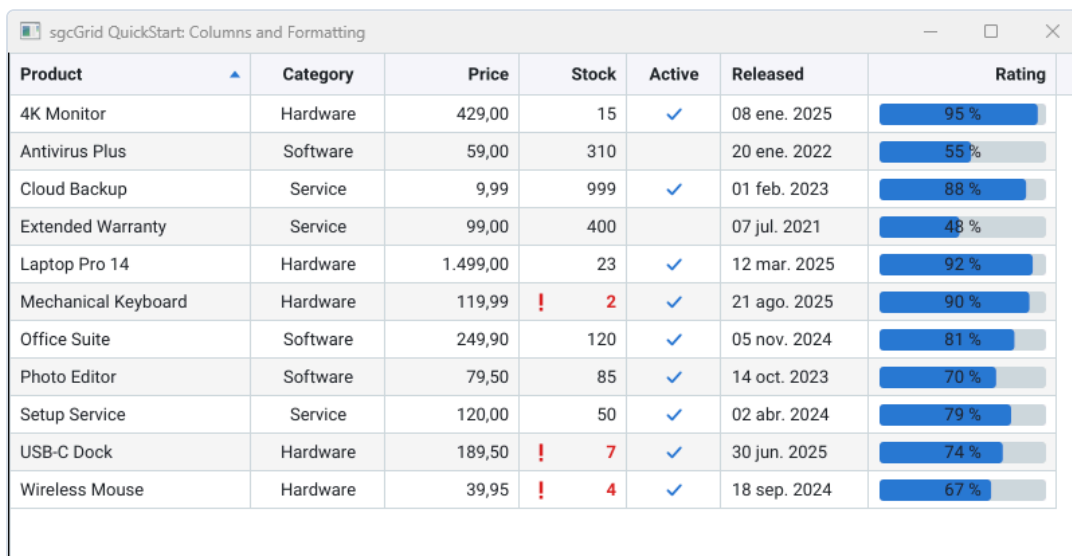
- Select the grid and set `Theme` to `utDark` in the Object Inspector. Run again.
- Set `RowHeader` to `[grhNumbers]` to show row numbers at the left of the rows. The other options of the set are `grhIndicator` and `grhCheckBox`.
- Add a fourth column with `FieldName` `capital`, a `Caption` of your choice and `DataType` `gdtBoolean`, then add a fourth value (`True` or `False`) to each `AppendRow` call. The new column shows check boxes.

4.2 Demo 02: Columns and Formatting

Folder `02.Columns`, project `GridColumns.dpr`, form `TfrmGridColumns` in `uGridColumns.pas`. Works in every edition.

What you will build

A product list where every column says what kind of value it holds and how to show it: text, currency, integers, a check box, dates and a progress bar. The product column stays in place when you scroll sideways, and a format rule paints low stock in red with a warning icon. Everything is set at design time; the code only adds the rows.



Product	Category	Price	Stock	Active	Released	Rating
4K Monitor	Hardware	429,00	15	✓	08 ene. 2025	95 %
Antivirus Plus	Software	59,00	310		20 ene. 2022	55 %
Cloud Backup	Service	9,99	999	✓	01 feb. 2023	88 %
Extended Warranty	Service	99,00	400		07 jul. 2021	48 %
Laptop Pro 14	Hardware	1.499,00	23	✓	12 mar. 2025	92 %
Mechanical Keyboard	Hardware	119,99	2	✓	21 ago. 2025	90 %
Office Suite	Software	249,90	120	✓	05 nov. 2024	81 %
Photo Editor	Software	79,50	85	✓	14 oct. 2023	70 %
Setup Service	Service	120,00	50	✓	02 abr. 2024	79 %
USB-C Dock	Hardware	189,50	7	✓	30 jun. 2025	74 %
Wireless Mouse	Hardware	39,95	4	✓	18 sep. 2024	67 %

Figure 4.2. Columns and Formatting. Three products have a stock under 10, so the format rule paints them red and bold with a warning icon. The Rating column is a progress bar.

Components on the form

Component	Class	Properties you set
Grid	TsgcVCLGrid	Align = <code>alClient</code> , FrozenColumns = 1, seven Columns (below) and one item in FormatRules

Column	DataType	Other properties	Result
product	(text)	SortOrder = <code>gsoAscending</code>	The grid starts sorted by product
category	(text)	Alignment = <code>gaCenter</code>	Centered text
price	<code>gdtCurrency</code>	DisplayFormat = <code>'#,##0.00'</code>	Two decimals, thousands separator
stock	<code>gdtInteger</code>	DisplayFormat = <code>'#,##0'</code>	Whole numbers, right aligned
active	<code>gdtBoolean</code>	Alignment = <code>gaCenter</code> , no DisplayFormat	A check box
released	<code>gdtDateTime</code>	DisplayFormat = <code>'dd mmm yyyy'</code>	A date with the short month name
rating	<code>gdtInteger</code>	DisplayFormat = <code>'0" %"'</code> , CellKind = <code>gcdProgress</code>	A progress bar with a percentage

The interesting parts of `uGridColumns.dfm` are the last column and the format rule:

```

// ...
item
  FieldName = 'rating'
  Caption = 'Rating'
  DataType = gdtInteger
  Width = 140
  DisplayFormat = '0" %"'
  CellKind = gcdProgress
end>
FrozenColumns = 1
FormatRules = <
  item
    FieldName = 'stock'
    Operator = gfoLess
    Value = '10'
    ForeColor = -2349530
    Bold = True
    Icon = gfiExclamation
    IconColor = -2349530
  end>
end

```

The colour `-2349530` looks odd, but it is just how the DFM stores the ARGB value `$FFDC2626`, an opaque red. In the Object Inspector you see and type the hexadecimal value; the DFM writes a Cardinal with the top bit set as a negative integer.

The code

A small helper keeps `FormCreate` readable. It passes each value with its real type: a string, a `Currency`, an `Integer`, a `Boolean`, a date and an integer rating.

```

procedure TfrmGridColumns.AddProduct(const aName, aCategory: string;
  aPrice: Currency; aStock: Integer; aActive: Boolean; aReleased: TDateTime;
  aRating: Integer);
begin
  { One value per column, in the order of the Columns collection. The
    values keep their real type (a number, a date, a Boolean), the
    DisplayFormat of each column turns them into text when painted. }
  Grid.Data.AppendRow(aName, [aName, aCategory, aPrice, aStock, aActive,
    VarFromDateTime(aReleased), aRating]);
end;

procedure TfrmGridColumns.FormCreate(Sender: TObject);
begin
  Grid.Data.BeginUpdate;
  try
    AddProduct('Laptop Pro 14', 'Hardware', 1499, 23, True,
      EncodeDate(2025, 3, 12), 92);
    AddProduct('Office Suite', 'Software', 249.9, 120, True,
      EncodeDate(2024, 11, 5), 81);
    // ...
    AddProduct('Extended Warranty', 'Service', 99, 400, False,
      EncodeDate(2021, 7, 7), 48);
  finally
    Grid.Data.EndUpdate;
  end;
end;

```

Note. The demos pass every date through `VarFromDateTime (unit Variants )`, which makes the value a date variant explicitly. Do the same in your own code, so a `gdtDateTime` column always receives a date, also when the value comes from date arithmetic such as `EncodeDate(2026, 1, 5) + i * 3` in demo 05.

How it works

1. When the form loads, the grid reads the seven columns. Because `product` has `SortOrder = gsoAscending`, the rows appear sorted by product as soon as they arrive.
2. Each value is stored as it was passed. Painting a cell turns it into text with the column's `DisplayFormat`: `FormatFloat` patterns for numbers, `FormatDateTime` patterns for dates.
3. The `active` column is `gdtBoolean` with an empty `DisplayFormat`, so it paints a check mark for `True` and nothing for `False`.
4. The `rating` column has `CellKind = gcdProgress`, so it paints a bar from `ProgressMin` to `ProgressMax` with the formatted value on top.
5. For every stock cell the format rule compares the value with `10`. When it is less, the cell gets the red text, the bold font and the exclamation icon.
6. `FrozenColumns = 1` keeps the product column in place. Make the window narrow and scroll right to see it.

Try this

- Double click `FormatRules`, select the rule and set `ApplyToRow` to `True`. The whole row of a low stock product is painted, not only the cell.
- Set the `DisplayFormat` of the `active` column to `Yes;No`. It becomes a text column showing Yes or No instead of a check box.
- Set the `ProgressColor` of the `rating` column to `$FF2563EB`, a blue. In code that is `Grid.Columns[6].ProgressColor := $FF2563EB;`
- Add a second rule in code at the end of `FormCreate`: `Grid.FormatRules.Add('stock', gfoGreater, '100', 0, $FF16A34A);` paints large stocks green. The parameters are the field name, the operator, the value, the background colour (0 keeps it) and the text colour. Add `sgcGrid_Format` to the `uses` clause for `gfoGreater`.

4.3 Demo 03: Editing and Events

Folder `03.Editing`, project `GridEditing.dpr`, form `TfrmGridEditing` in `uGridEditing.pas`.

Watch out. This demo does not compile with the Community edition. In-place editing and the events `OnValidate`, `OnEdited` and `OnEditError` are not part of Community, so the compiler stops with *Undeclared identifier*, and a form that uses them cannot be loaded. Use a Trial or Registered installation to run it.

What you will build

A small task list that the user edits in place. Owner and priority are chosen from drop-down lists, the code column cannot be changed, the done column toggles with a click, and every change, refusal and typing error is written to a log under the grid.

Select a cell and type, or press F2, Enter or double click. Enter or Tab saves, Escape cancels. Click a check box to toggle it.

Code	Task	Owner	Priority	Hours	Due	Done
T-101	Write the user manual	Anna	High	12,5	2026-03-02	
T-102	Fix the login bug	Peter	High	3,0	2026-02-20	✓
T-103	Design the new logo	Laura	Medium	8,0	2026-03-15	
T-104	Update the price list	Anna	Low	1,5	2026-02-27	✓
T-105	Prepare the trade fair	Mark	Medium	20,0	2026-04-10	
T-106	Answer support tickets	Peter	Low	6,0	2026-03-06	

Ready. Select a cell and type to change it.

Figure 4.3. Editing and Events, before any change. The help line at the top lists the keys; the memo at the bottom receives the log.

Components on the form

Component	Class	Properties you set
lblHelp	TLabel	Align = alTop, the keyboard help line
Grid	TsgcVCLGrid	Align = alClient, ReadOnly = False, SelectionMode = gsmCell, OnValidate = GridValidate, OnEdited = GridEdited, OnEditError = GridEditError
memLog	TMemo	Align = alBottom, Height = 120, ReadOnly = True

The columns are `code` (`Editable = False`), `task`, `owner` and `priority` (each with a `PickList`), `hours` (`gdtFloat`, `'0.0'`), `due` (`gdtDateTime`, `'yyyy-mm-dd'`) and `done` (`gdtBoolean`, a check box). From `uGridEditing.dfm`:

```

Columns = <
  item
    FieldName = 'code'
    Caption = 'Code'
    Width = 70
    Editable = False
  end
  // ...
  item
    FieldName = 'owner'
    Caption = 'Owner'
    PickList.Strings = (
      'Anna'
      'Laura'
      'Mark'
      'Peter')
  end
  // ...
SelectionMode = gsmCell
ReadOnly = False
FormatRules = <>
OnValidate = GridValidate
OnEdited = GridEdited
OnEditError = GridEditError
end

```

`SelectionMode = gsmCell` moves a cell cursor instead of selecting whole rows, which is what users expect when they edit cell by cell.

The code

`FormCreate` adds six tasks with `AppendRow`, exactly as in the previous demos. The three event handlers are the new part. `OnValidate` receives the new value, already converted to the type of the column, and decides whether to accept it:

```

procedure TfrmGridEditing.GridValidate(Sender: TObject; aRow, aCol: Integer;
  var aValue: Variant; var aAccept: Boolean);
var
  vField: string;
begin
  { aCol is the index in Grid.Columns, the FieldName tells which column it
    is without depending on the column order. }
  vField := Grid.Columns[aCol].FieldName;
  if (vField = 'task') and (Trim(VarToStr(aValue)) = '') then
    begin
      aAccept := False;
      Log('Refused: the task name cannot be empty.');
    end
    else if (vField = 'hours') and not VarIsNull(aValue) and
      ((aValue < 0) or (aValue > 100)) then
      begin
        aAccept := False;
        Log(Format('Refused: %s hours, the value must be between 0 and 100.',
          [VarToStr(aValue)]));
      end;
    end;
end;

```

`OnEditError` fires before validation when the typed text cannot be converted at all, and `OnEdited` fires after a value was stored:

```

procedure TfrmGridEditing.GridEditError(Sender: TObject; aRow, aCol: Integer;
  const aText: string; var aMessage: string);
begin
  { aMessage is the text the grid shows to the user, it can be changed. }
  Log(Format('Invalid input "%s" in %s: %s', [aText,
    Grid.Columns[aCol].Caption, aMessage]));
end;

procedure TfrmGridEditing.GridEdited(Sender: TObject; aRow, aCol: Integer;
  const aOldValue, aNewValue: Variant);
begin
  { Grid.Data.KeyOf returns the key given to AppendRow for that row. }
  Log(Format('%s, %s changed from "%s" to "%s"', [Grid.Data.KeyOf(aRow),
    Grid.Columns[aCol].Caption, VarToStr(aOldValue), VarToStr(aNewValue)]));
end;

```

`Log` is a one line private method that adds the text to `memLog.Lines`.

How it works

1. `ReadOnly = False` turns editing on for the whole grid; `Editable = False` turns it off again for the code column.
2. The user selects a cell and types, or presses F2 or Enter, or double clicks. The grid opens the editor of the column over the cell: a text box, a combo box for a column with a `PickList`, or a date picker for the `gdtDateTime` column.
3. Enter or Tab saves, Escape cancels. A click or the space bar toggles a `gdtBoolean` cell directly, without an editor.
4. If the text cannot be converted to the column type, for example `abc` in Hours, the grid raises `OnEditError` and keeps the editor open, so the user can correct the text.
5. Otherwise it raises `OnValidate` with the converted value. Setting `aAccept := False` refuses it.
6. An accepted value is stored and `OnEdited` reports the old and the new value. `Grid.Data.KeyOf(aRow)` turns the view row into the task code, which stays correct even after the user sorts.

Try this

- Set the `EditorKind` of the `hours` column to `gedSpin` to edit hours with a spin editor. Other values include `gedDate`, `gedTime`, `gedColor` and `gedMemo`.
- Add an `OnBeforeEdit` handler, `(Sender: TObject; aRow, aCol: Integer; var aAllow: Boolean)`, with the line `aAllow := Grid.Data.FieldValue(aRow, 'done') <> True`; so that finished tasks cannot be changed.
- Set `SelectionMode` to `gsmCellRange` for spreadsheet style selection with copy and paste, or set `AllowInsert` to `True` to let the user add rows from the keyboard. The full `Editing` demo shows both.

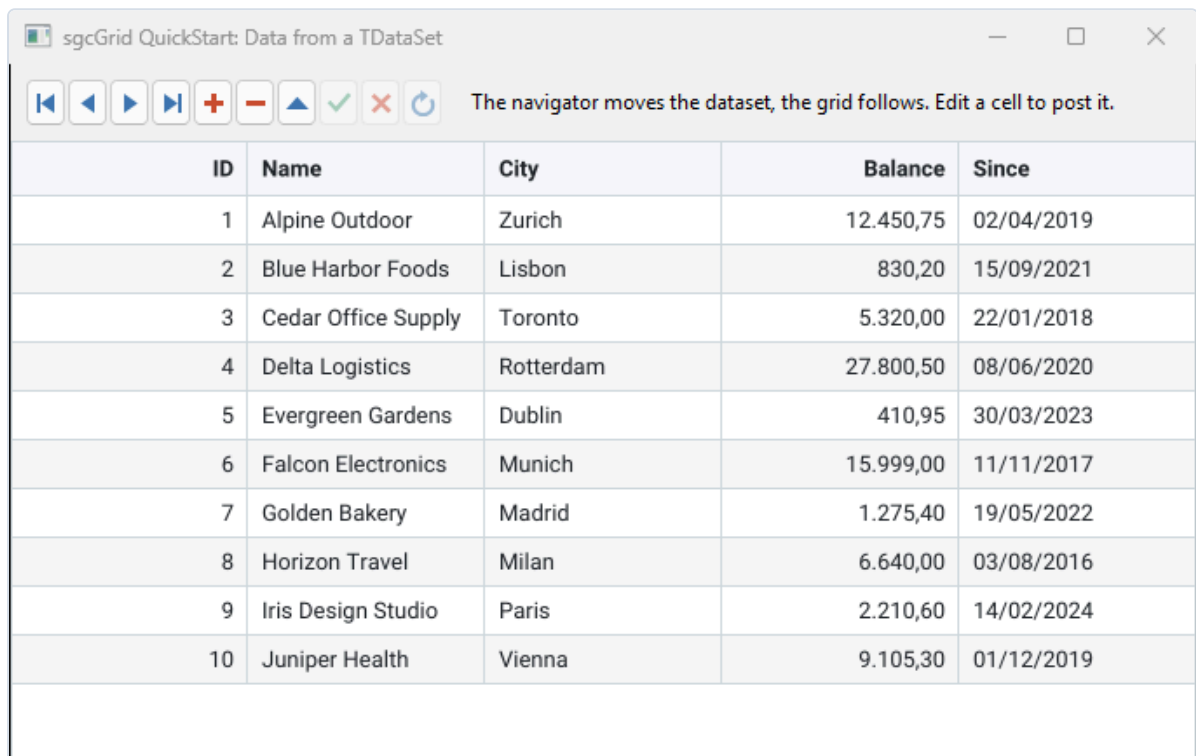
4.4 Demo 04: DataSet

Folder `04.DataSet` , project `GridDataSet.dpr` , form `TfrmGridDataSet` in `uGridDataSet.pas` .

Watch out. This demo does not compile with the Community edition, because `SyncCursor` and `WriteBack` of `TsgcGridSource_DataSet` are not part of Community. The source itself is: with Community, a grid can load and follow a dataset, as described under *How it works*, but it cannot write edits back or follow the current record.

What you will build

A customer list that lives in a `TClientDataSet` and is shown in the grid. A `TDBNavigator` moves through the dataset and the selected grid row follows it; selecting a row in the grid moves the dataset. Editing a cell posts the change to the dataset. The grid columns are created from the dataset fields, so the grid has no columns at design time.



The screenshot shows a window titled "sgcGrid QuickStart: Data from a TDataSet". Below the title bar is a toolbar with navigation icons (back, forward, first, last, insert, delete, up, down, save, cancel, refresh) and a text label: "The navigator moves the dataset, the grid follows. Edit a cell to post it." Below the toolbar is a grid with 5 columns: ID, Name, City, Balance, and Since. The grid contains 10 rows of customer data.

ID	Name	City	Balance	Since
1	Alpine Outdoor	Zurich	12.450,75	02/04/2019
2	Blue Harbor Foods	Lisbon	830,20	15/09/2021
3	Cedar Office Supply	Toronto	5.320,00	22/01/2018
4	Delta Logistics	Rotterdam	27.800,50	08/06/2020
5	Evergreen Gardens	Dublin	410,95	30/03/2023
6	Falcon Electronics	Munich	15.999,00	11/11/2017
7	Golden Bakery	Madrid	1.275,40	19/05/2022
8	Horizon Travel	Milan	6.640,00	03/08/2016
9	Iris Design Studio	Paris	2.210,60	14/02/2024
10	Juniper Health	Vienna	9.105,30	01/12/2019

Figure 4.4. DataSet. The five columns were created from the five fields of the dataset, with the field names as captions.

Components on the form

Component	Class	Properties you set
<code>cdsCustomers</code>	<code>TClientDataSet</code>	Nothing: the fields are created in code
<code>dsCustomers</code>	<code>TDataSource</code>	<code>DataSet = cdsCustomers</code> . Only the navigator uses it
<code>pnlTop</code>	<code>TPanel</code>	<code>Align = alTop</code> , holds the navigator and <code>lblHelp</code>
<code>navCustomers</code>	<code>TDBNavigator</code>	<code>DataSource = dsCustomers</code>
<code>GridSource</code>	<code>TsgcGridSource_DataSet</code>	<code>DataSet = cdsCustomers</code> , <code>Grid = Grid</code> , <code>KeyField = 'ID'</code> , <code>SyncCursor = True</code> , <code>WriteBack = True</code>
<code>Grid</code>	<code>TsgcVCLGrid</code>	<code>Align = alClient</code> , <code>NO Columns</code> , <code>ReadOnly = False</code> , <code>AutoFitColumns = True</code>

The bridge between the dataset and the grid is `GridSource`. Note that the grid does not use the `TDataSource`: the source component talks to the dataset directly. From `uGridDataSet.dfm`:

```
object Grid: TsgcVCLGrid
    // ...
    Align = alClient
    TabOrder = 1
    Columns = <>
    ReadOnly = False
    FormatRules = <>
    AutoFitColumns = True
end
// ...
object GridSource: TsgcGridSource_DataSet
    DataSet = cdsCustomers
    Grid = Grid
    KeyField = 'ID'
    SyncCursor = True
    WriteBack = True
    Left = 256
    Top = 320
end
```

Two more properties of the source matter here, and both keep their default value, so the DFM does not mention them: `AutoColumns (True)` creates the columns from the fields, and `LiveUpdates (True)` makes the grid follow every change posted to the dataset.

The code

The `uses` clause has the database units, `MidasLib` so the `TClientDataSet` needs no `midas.dll`, and `sgcGrid_Source_DataSet` for the source component:

```
uses
    Classes, Controls, Forms, StdCtrls, ExtCtrls, SysUtils, DB, DBClient,
    DBCtrls, MidasLib,
    // sgc
    sgcGrid_VCL, sgcGrid_Source_DataSet;
```

`CreateCustomers` builds an in-memory table so the demo needs no database. In your application this is where a query opens:

```

procedure TfrmGridDataSet.CreateCustomers;

  procedure AddCustomer(aID: Integer; const aName, aCity: string;
    aBalance: Currency; aSince: TDateTime);
  begin
    cdsCustomers.Append;
    cdsCustomers.FieldByName('ID').AsInteger := aID;
    cdsCustomers.FieldByName('Name').AsString := aName;
    cdsCustomers.FieldByName('City').AsString := aCity;
    cdsCustomers.FieldByName('Balance').AsCurrency := aBalance;
    cdsCustomers.FieldByName('Since').AsDateTime := aSince;
    cdsCustomers.Post;
  end;

begin
  { The fields of the dataset. With a database query they come from the
    SELECT statement instead. }
  cdsCustomers.FieldDefs.Add('ID', ftInteger);
  cdsCustomers.FieldDefs.Add('Name', ftString, 40);
  cdsCustomers.FieldDefs.Add('City', ftString, 30);
  cdsCustomers.FieldDefs.Add('Balance', ftCurrency);
  cdsCustomers.FieldDefs.Add('Since', ftDate);
  cdsCustomers.CreateDataSet;

  AddCustomer(1, 'Alpine Outdoor', 'Zurich', 12450.75, EncodeDate(2019, 4, 2));
  AddCustomer(2, 'Blue Harbor Foods', 'Lisbon', 830.2, EncodeDate(2021, 9, 15));
  // ...
  AddCustomer(10, 'Juniper Health', 'Vienna', 9105.3, EncodeDate(2019, 12, 1));
  cdsCustomers.First;
end;

```

And the only line that concerns the grid:

```

procedure TfrmGridDataSet.FormCreate(Sender: TObject);
begin
  CreateCustomers;
  { Loads every record into the grid. From now on the grid and the dataset
    stay in step on their own, no more code is needed. }
  GridSource.Active := True;
end;

```

How it works

1. `GridSource.Active := True` walks the dataset (with `DisableControls`, restoring the current record afterwards) and copies every record into the grid.
2. The grid has no columns, so `AutoColumns` creates one per field. The caption is the field's `DisplayLabel` and the data type follows the field type: the `ftCurrency` field becomes a currency column, the `ftDate` field a date column.
3. `KeyField = 'ID'` gives every row the ID as its key. That is what lets the source update a row in place instead of reloading the grid.
4. With `LiveUpdates`, a change posted to the dataset, for example with the navigator's delete or refresh buttons, is reflected in the grid.
5. `SyncCursor` links the selected grid row and the current record in both directions: the navigator moves the grid, a click in the grid moves the dataset.
6. `WriteBack` writes a grid edit to the dataset and posts it. If the post is refused, the grid keeps the old value and raises `OnEditError`.

Note. The source loads every record into the grid, which then sorts, filters and groups in memory. That is the right choice for the tables most screens show. For a table too large to load, the Registered edition has server mode, where `TsgcGridServerProvider_DataSet` reads one page at a time and lets the database sort, filter and group.

Try this

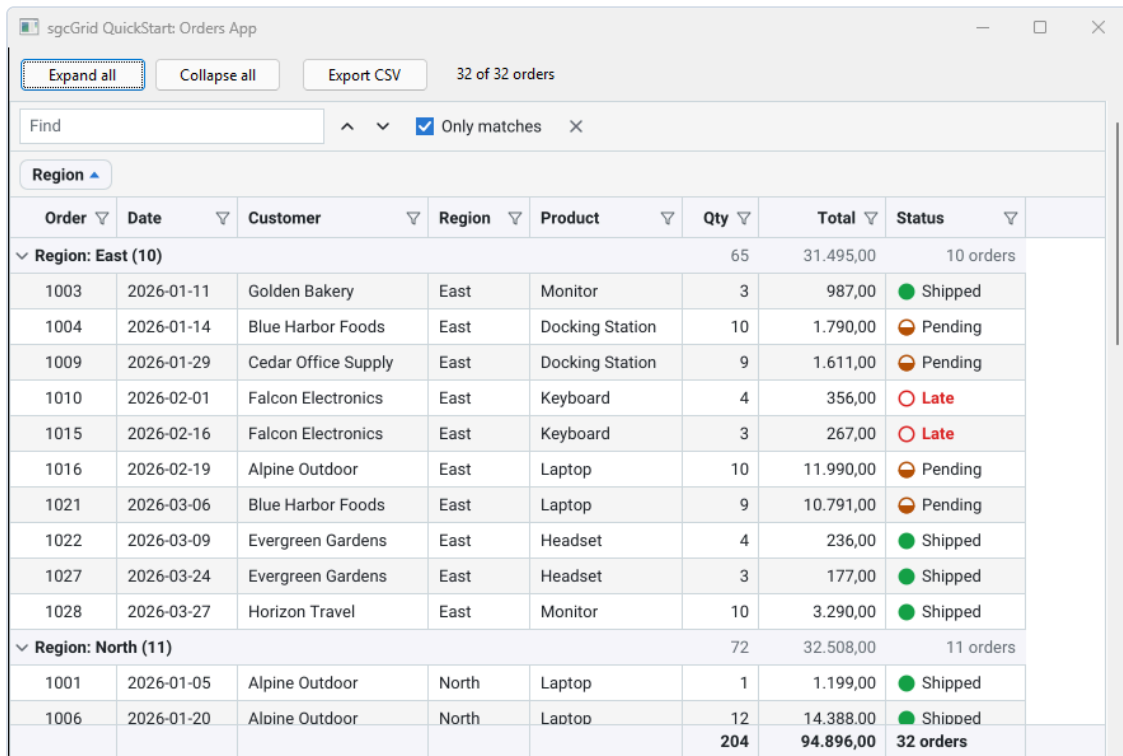
- Add a design time column to the grid with `FieldName` `Balance` and `DisplayFormat` `#,##0.00 "EUR"`. The source keeps your column, matches it to the field by name and lets `AutoColumns` create the others.
- Set `ApplyFilter` to `True` on `GridSource` and `ShowFindPanel` to `True` on the grid. Type in the find box with **Only matches** checked: the dataset then holds only the records of the rows the grid shows, and the navigator moves through those. This needs a `KeyField`.
- Replace the `TClientDataSet` by a `TFDQuery` or any other `TDataSet`. Only `GridSource.DataSet` (and the navigator's data source) change.

4.5 Demo 05: Orders App

Folder `05.OrdersApp`, project `GridOrders.dpr`, form `TfrmGridOrders` in `uGridOrders.pas`. Works in every edition.

What you will build

A small but realistic screen: 32 orders grouped by region, with quantity and total summed per region and in the footer, an order count, a search box, a filter button in every header, a coloured status icon, buttons to expand and collapse the groups, a label that counts the visible orders, and an export to CSV. Almost all of it is set in the Object Inspector.



Order	Date	Customer	Region	Product	Qty	Total	Status
Region: East (10)					65	31.495,00	10 orders
1003	2026-01-11	Golden Bakery	East	Monitor	3	987,00	Shipped
1004	2026-01-14	Blue Harbor Foods	East	Docking Station	10	1.790,00	Pending
1009	2026-01-29	Cedar Office Supply	East	Docking Station	9	1.611,00	Pending
1010	2026-02-01	Falcon Electronics	East	Keyboard	4	356,00	Late
1015	2026-02-16	Falcon Electronics	East	Keyboard	3	267,00	Late
1016	2026-02-19	Alpine Outdoor	East	Laptop	10	11.990,00	Pending
1021	2026-03-06	Blue Harbor Foods	East	Laptop	9	10.791,00	Pending
1022	2026-03-09	Evergreen Gardens	East	Headset	4	236,00	Shipped
1027	2026-03-24	Evergreen Gardens	East	Headset	3	177,00	Shipped
1028	2026-03-27	Horizon Travel	East	Monitor	10	3.290,00	Shipped
Region: North (11)					72	32.508,00	11 orders
1001	2026-01-05	Alpine Outdoor	North	Laptop	1	1.199,00	Shipped
1006	2026-01-20	Alpine Outdoor	North	Laptop	12	14.388,00	Shipped
					204	94.896,00	32 orders

Figure 4.5. Orders App. The group row of each region shows its own summaries, and the footer shows the totals of all visible orders.

Components on the form

Component	Class	Properties you set
<code>pnlTop</code>	<code>TPanel</code>	<code>Align = alTop</code> , holds the buttons and the label
<code>btnExpand</code> , <code>btnCollapse</code> , <code>btnExport</code>	<code>TButton</code>	Captions Expand all, Collapse all and Export CSV, each with an <code>onClick</code>
<code>lblCount</code>	<code>TLabel</code>	Shows how many orders are visible
Grid	<code>TsgcVCLGrid</code>	<code>Align = alClient</code> , <code>ShowFooter</code> , <code>ShowGroupPanel</code> , <code>ShowFilterButtons</code> , <code>ShowFindPanel</code> and <code>FindOnlyMatches</code> all <code>True</code> , <code>OnViewChange = GridViewChange</code> , eight <code>Columns</code> and three <code>FormatRules</code>

The columns carry the grouping and the summaries. `region` has `GroupIndex = 0`, which groups by it from the start; `qty` and `total` are summed, and `status` is counted with its own format. From `uGridOrders.dfm`:

```

item
  FieldName = 'region'
  Caption = 'Region'
  Width = 80
  GroupIndex = 0
end
// ...
item
  FieldName = 'total'
  Caption = 'Total'
  DataType = gdtCurrency
  Width = 100
  DisplayFormat = '#,##0.00'
  SummaryType = gsuSum
end
item
  FieldName = 'status'
  Caption = 'Status'
  Width = 110
  SummaryType = gsuCount
  SummaryFormat = '#,##0 "orders"'
end>

```

The three format rules have no `Operator` in the DFM because they use the default, `gfoEqual`. Each one adds an icon for one status, and the last one also paints late orders red and bold:

```

FormatRules = <
  item
    FieldName = 'status'
    Value = 'Shipped'
    Icon = gfiCircleGreen
    IconColor = -15293622
  end
  item
    FieldName = 'status'
    Value = 'Pending'
    Icon = gfiCircleAmber
    IconColor = -4959479
  end
  item
    FieldName = 'status'
    Value = 'Late'
    ForeColor = -2349530
    Bold = True
    Icon = gfiCircleRed
    IconColor = -2349530
  end
end>

```

The code

The orders are built from a few constant arrays, so every run shows the same screen. `sgcGrid_I0` in the `uses` clause provides the CSV export.

```

procedure TfrmGridOrders.AddOrders;
var
  i, vProduct, vQty: Integer;
begin
  { 32 orders made from the lists above, always the same ones so every run
    shows the same screen. }
  Grid.Data.BeginUpdate;
  try
    for i := 0 to 31 do
    begin
      vProduct := (i * 3) mod 5;
      vQty := 1 + (i * 7) mod 12;
      { AppendRow(Key, [Order, Date, Customer, Region, Product, Qty, Total,
        Status]) }
      Grid.Data.AppendRow(IntToStr(1001 + i), [1001 + i,
        VarFromDateTime(EncodeDate(2026, 1, 5) + i * 3),
        CS_CUSTOMERS[(i * 3 + i div 4) mod 8],
        CS_REGIONS[(i * 7 + i div 3) mod 4], CS_PRODUCTS[vProduct], vQty,
        vQty * CS_PRICES[vProduct], CS_STATUS[(i * 4 + i div 3) mod 7]]);
    end;
  finally
    Grid.Data.EndUpdate;
  end;
end;

```

The rest is four short handlers. `OnViewChange` keeps the label up to date, and the buttons call one method each:

```

procedure TfrmGridOrders.FormCreate(Sender: TObject);
begin
  AddOrders;
  GridViewChange(nil);
end;

procedure TfrmGridOrders.GridViewChange(Sender: TObject);
begin
  { OnViewChange fires after a search, a filter or new rows. ViewCount is
    the number of rows that pass the search and the filters, RowCount the
    number of rows stored in the grid. }
  lblCount.Caption := Format('%d of %d orders', [Grid.Data.ViewCount,
    Grid.Data.RowCount]);
end;

procedure TfrmGridOrders.btnExpandClick(Sender: TObject);
begin
  Grid.ExpandAll;
end;

procedure TfrmGridOrders.btnCollapseClick(Sender: TObject);
begin
  Grid.CollapseAll;
end;

```

```

procedure TfrmGridOrders.btnExportClick(Sender: TObject);
var
    vFileName: string;
begin
    { GridSaveToCSVFile (unit sgcGrid_IO) writes the rows in the order shown,
      without the rows hidden by the search or the filters. True writes only
      the visible columns. }
    vFileName := ExtractFilePath(ParamStr(0)) + 'orders.csv';
    GridSaveToCSVFile(Grid.Data, vFileName, True);
    lblCount.Caption := 'Saved to ' + vFileName;
end;

```

How it works

1. `AddOrders` stores 32 rows. Because the region column has `GroupIndex = 0`, the grid shows one group row per region, with the number of orders in brackets, and the orders of that region under it.
2. With `ShowFooter` the footer shows the summaries of the columns that have a `SummaryType`: the sum of quantity and total, and the count of orders formatted by `SummaryFormat`. Each group row repeats them for its own orders.
3. The group bar above the header (`ShowGroupPanel`) shows Region. Drag another header onto it to add a second level, or drag Region back to the header row to ungroup.
4. Typing in the find box (`ShowFindPanel`; Ctrl+F focuses it) highlights the matching text in the cells. With `FindOnlyMatches` only the matching rows stay visible.
5. The funnel in each header (`ShowFilterButtons`) opens a list of the values of that column with their counts, to filter by one or more of them.
6. Every search or filter changes the view, so the grid raises `OnViewChange` and the label shows how many of the 32 orders are visible.
7. **Export CSV** writes the visible columns of the visible rows, in the order shown, next to the executable. Numbers use a dot as the decimal separator and dates are written as ISO dates by default, so the file reads the same on every machine.

Try this

- Set `ShowFilterRow` to `True`: a row under the header lets you type a filter per column.
- Set `GroupsExpanded` to `False`: the screen starts with the four regions collapsed, each with its totals.
- Export to Excel instead of CSV: in `btnExportClick` write `Grid.SaveToXLSX(ExtractFilePath(ParamStr(0)) + 'orders.xlsx');`. The workbook keeps the groups as an outline and the summaries. Excel export is not part of the Community edition.
- Set `SelectionMode` to `gsmMultiRow` and `SummaryScope` to `gssSelection`: the footer then sums only the rows you select.

5. Main components

The QuickStart uses two components, `TsgcVCLGrid` and `TsgcGridSource_DataSet`, and that is enough for most screens. This chapter introduces the rest of the SGC Grid palette page, plus the classes you meet in code, so you know what is there when you need it. Every class has its own help topic, named after the class, that lists every property with its type and default value.

Reference table

Group	Class	Palette	What it is for
Grid	<code>TsgcVCLGrid</code>	SGC Grid (VCL)	The VCL grid control
Grid	<code>TsgcFMXGrid</code>	SGC Grid (FireMonkey)	The FireMonkey grid control, with the same properties, methods and events
Code	<code>TsgcGridData</code>	none, <code>Grid.Data</code>	The rows: append, update, delete by key, sort, filter, count; thread-safe
Code	<code>TsgcGridColumn</code>	none, <code>Grid.Columns</code>	One column: field name, caption, data type, format, cell kind, summary
Code	<code>TsgcGridFormatRule</code>	none, <code>Grid.FormatRules</code>	One conditional format: colours, bold, icon, whole row
Data	<code>TsgcGridSource_DataSet</code>	SGC Grid	Fills a grid from any <code>TDataSet</code> and keeps it in step
Data	<code>TsgcGridDataStore</code>	SGC Grid	Rows without a screen, the source of detail bands. Not in Community
Data	<code>TsgcGridServerProvider_DataSet</code>	SGC Grid	Server mode over an SQL query: the database pages, sorts, filters and groups. Not in Community
Data	<code>TsgcGridServerProvider_Callback</code>	SGC Grid	Server mode where your code answers the page, sort and filter requests. Not in Community
Alarms	<code>TsgcGridAlarmModel</code>	SGC Grid	An ISA-18.2 alarm list: states, priorities, acknowledge, shelf
Alarms	<code>TsgcGridAlarmLink</code>	SGC Grid	Shows an alarm model in a grid
Alarms	<code>TsgcVCLGridAlarmMenu</code> , <code>TsgcFMXGridAlarmMenu</code>	SGC Grid	The operator menu of the alarm list
Log	<code>TsgcGridLogModel</code>	SGC Grid	A log in a ring buffer, with levels, a filter and exports
Log	<code>TsgcGridLogLink</code>	SGC Grid	Shows a log model in a grid
AI	<code>TsgcGridAI</code>	SGC Grid	Turns a question in plain language into filters, sorts and highlights
Bridge	<code>TsgcGridFeedSource_WebSocket</code> , <code>_MQTT</code> , <code>_SSE</code>	SGC Grid (sgcGridWS)	Write the JSON rows of a live feed into a grid by key. Need <code>sgcWebSockets</code>

Group	Class	Palette	What it is for
Bridge	<code>TsgcGridAIQuery</code>	SGC Grid (sgcGridWS)	The classic AI query component of the bridge

The grid controls

TsgcVCLGrid

The control you have used in every demo. It owns a `TsgcGridData` (`Data`), the `Columns` and the `FormatRules`, and adds the painting, the scroll bars, the mouse, the keyboard and the built in features: sorting, frozen columns, the row header, the find panel, the filter buttons and filter row, the group panel, the footer and, outside Community, in-place editing and Excel export. Its look comes from `Theme`, `AccentColor`, `FontSize`, `RowHeight` and `HeaderHeight`. Right click it in the designer for the **Columns editor...** and **Import columns from DB grid...** verbs. Help: `TsgcVCLGrid.htm`.

TsgcFMXGrid

The FireMonkey grid, from Delphi XE6, for Windows, macOS, iOS and Android. It has the same properties, methods and events as the VCL grid, with drawn scroll bars and touch panning, so the code of this guide works with it unchanged. The demos whose folder name ends in `FMX` use it. Help: `TsgcFMXGrid.htm`.

Data and sources

TsgcGridData, TsgcGridColumn and TsgcGridFormatRule

These are not on the palette; you reach them through the grid. `TsgcGridData` is where your code spends most of its time: `AppendRow`, `UpdateRow`, `UpdateCell`, `DeleteRow`, `Clear`, `BeginUpdate` and `EndUpdate`, `RowCount` and `ViewCount`, `KeyOf` and `FieldValue`. `TsgcGridColumn` and `TsgcGridFormatRule` are the items of the `Columns` and `FormatRules` collections described in chapter 3. Help: `TsgcGridData.htm`, `TsgcGridColumn.htm`, `TsgcGridFormatRule.htm`.

TsgcGridSource_DataSet

The bridge from any `TDataSet`, as in demo 04. Set `DataSet`, `Grid` and `KeyField`, then `Active`. `AutoColumns` creates columns from the fields, `LiveUpdates` follows the changes of the dataset, `BooleanFields` turns fields with values such as S and N into check columns, and `ApplyFilter` filters the dataset to the rows the grid shows. Outside Community, `SyncCursor` links the current record and the selected row and `WriteBack` posts grid edits, inserts and deletes. Help: `TsgcGridSource_DataSet.htm`.

TsgcGridDataStore and the server providers

`TsgcGridDataStore` holds rows without showing them. Its job is to be the source of the detail bands of a master detail grid, filled by a `TsgcGridSource_DataSet` or by code. `TsgcGridServerProvider_DataSet` and `TsgcGridServerProvider_Callback` are for server mode, where the grid never loads the whole table: it asks for one page at a time and the database (or your callback) sorts, filters, counts, summarizes and groups. The provider for SQL works with any query component, such as `TFDQuery`. These three are not in the Community edition. Help: `TsgcGridDataStore.htm`, `TsgcGridServerProvider_DataSet.htm`.

Alarm list and log viewer

Two ready-made models turn a grid into a specialised viewer. `TsgcGridAlarmModel` keeps an ISA-18.2 alarm list: your application raises and clears alarms, and the model keeps their state and takes the operator commands (acknowledge, shelve and others). `TsgcGridAlarmLink` shows the model in a grid, one row per alarm, and the alarm menu components give the operator the commands. `TsgcGridLogModel` keeps the newest lines of a log in a ring buffer, with levels, a filtered view, counters and exports to text, JSON and CSV; lines can be added from any thread. `TsgcGridLogLink` shows it in a grid. Help: `TsgcGridAlarmModel.htm`, `TsgcGridLogModel.htm` and the *Alarm List* and *Log Viewer* topics.

Asking the grid in plain language

`TsgcGridAI` turns a question such as *price greater than 100 sort by volume descending* into filters, sorts and highlights of a grid. It asks the AI provider of your choice, or answers with its own rule parser when no provider is assigned, so it works without an API key. It is in every edition, Community included. The full demo `AskYourGrid` shows it with a query bar. Help: `TsgcGridAI.htm` and the *Ask Your Grid* topic.

The sgcWebSockets bridge

When `sgcWebSockets` is installed, the optional **sgcGrid for sgcWebSockets** component of the installer adds the `sgcGridWS` packages. Their feed sources, `TsgcGridFeedSource_WebSocket`, `TsgcGridFeedSource_MQTT` and `TsgcGridFeedSource_SSE`, read every message of a live feed as JSON and write its rows into a grid by key: only the fields present in a message are written, so a message can carry a partial update. The full demo `LiveFeed` shows a grid fed by a WebSocket stream with no code in between. The bridge is not part of the Community edition. Help: `TsgcGridFeedSource.htm` and the *Live Feeds* topic.

6. Common problems

These are the problems people actually run into in their first days with `sgcGrid`, with the fix for each one.

Compiling and editions

Undeclared identifier OnValidate, OnEdited, SyncCursor or WriteBack

You are compiling with the Community edition. It is built without in-place editing (the define `SGC_GRID_EDIT`), so the editing members of the grid and the `SyncCursor` and `WriteBack` properties of the `DataSet` source do not exist, and code that uses them stops with *E2003 Undeclared identifier*. This is why demos 03 and 04 do not compile with Community. Remove the editing code, or use a Trial or Registered installation.

Error reading a form when it loads

A form saved by a Registered or Trial installation with editing events or properties, or with tree or detail settings, cannot be loaded by a Community build, which raises an *Error reading* exception for the unknown property. Open the form in a full edition and remove those settings, or keep such forms out of Community projects.

Undeclared identifier SaveToXLSX, Tree or Detail

Excel export, tree rows, master detail rows and `TsgcGridDataStore` are compiled only with `SGC_GRID_ADVANCED`, which the Community edition leaves out. Export to CSV or JSON with the functions of `sgcGrid_IO` instead, as demo 05 does.

File not found: sgcGridWS_Source or sgcWebSocket

The feed sources and `TsgcGridAIQuery` are in the `sgcGridWS` bridge. The Community edition does not ship it. In the other editions the bridge needs `sgcWebSockets`: install the `sgcWebSockets` package before the `sgcGridWS` package of the same Delphi version, or add the `sgcWebSockets` sources to the search path.

Rows and values

Values do not change, or rows are duplicated

`UpdateRow` finds rows by key, and keys are case-sensitive. If you appended a row as `'MSFT'` and update `'msft'`, you get a second row (or nothing, when `Data.AutoInsert` is `False`). Use exactly the same key for `AppendRow` and `UpdateRow`. Also check the field names: a field name that does not match the `FieldName` of a column is ignored without an error.

Values land in the wrong column

`AppendRow` takes the values in the order of the `Columns` collection, not by name. If you reorder or add columns in the Columns editor, update your `AppendRow` calls, or use `UpdateRow`, which takes field names.

Numbers show a comma or a dot unexpectedly

Values are formatted with `Data.FormatSettings`, which follow the regional settings of the machine. That is why the screenshots in this guide show `1.499,00`. Call `Grid.Data.UseInvariantFormat` for a dot as the decimal separator and ISO dates, or set the fields of `Data.FormatSettings` yourself.

The user cannot edit a cell

A grid is read-only until you set `ReadOnly` to `False`, and then a column with `Editable = False` stays read-only, like the code column in demo 03. An `OnBeforeEdit` handler that sets `aAllow` to `False` also blocks the editor. In the Community edition there is no in-place editing at all.

OnFilterRow reads the wrong row

`OnFilterRow` receives a *store* row, the row in the order it was added, because it runs while the view is being built. Read it with `StoreCells` or `StoreKeyOf`, not with `Cells`, which takes view rows.

Threads

The grid freezes when a handler touches a control

`OnFilterRow` and `OnGetValue` run with the data lock held, and `OnChange` of the data and the events of the feed sources run on the writer thread. Never wait for the main thread there and never touch a control. Use `TThread.Queue` to hand the work to the main thread.

A worker thread outlives the form

A worker thread that writes to `Grid.Data` must be stopped before the form frees the grid. Terminate it and wait for it in `FormDestroy` or in the destructor of the form, as the thread example in chapter 3 does.

DataSets

The whole grid reloads on every change, or `ApplyFilter` does nothing

Without a `KeyField` the source cannot match records to rows, and `ApplyFilter` is ignored. Set `KeyField` to a field whose values are unique, such as the primary key. Then changed rows are updated in place (and flash), new keys are appended, missing keys are deleted, and `ApplyFilter` works.

The application asks for `midas.dll`

A `TClientDataSet` needs the MIDAS library. Demo 04 has `MidasLib` in its `uses` clause, which links it into the executable. Keep that unit, or ship `midas.dll` with your application.

The columns are not the ones I designed

When the grid has no columns, `AutoColumns` creates one per field. When you design columns yourself, the source keeps them and matches them to the fields by `FieldName`, so check that each `FieldName` is spelled like the field. Set `AutoColumns` to `False` if you never want columns created for you.

7. Where next

Once the five QuickStart demos feel familiar, you know the core of sgcGrid. This chapter points you to the larger demos, the help and the website.

The full demos

Every folder in `Demos\Grid\Delphi` next to `00.QuickStart` is a complete demo of one area, with its own `readme.html`. The folders whose name ends in `FMX` are FireMonkey versions of the demo with the same name.

Demo	What it shows
<code>AskYourGrid</code>	Filter, sort and highlight with questions in plain language, no API key needed.
<code>BandsAndCells</code>	Band headers, sparkline, progress and image cells, and a multi-row selection that survives sorting and live updates.
<code>CardTemplateFMX</code>	Card templates, detail cards and swipe actions on a phone.
<code>DataSetEditing</code>	Editing a <code>TClientDataSet</code> through the grid: write back, a lookup column, undo and redo.
<code>DataSetGrid</code>	A grid bound to a <code>TClientDataSet</code> , plus JSON and CSV export and import.
<code>Editing</code>	In-place editors, validation, paste and undo.
<code>ExcelExport</code>	A grouped grid and a tree grid saved as Excel workbooks.
<code>FilterAndFind</code> , <code>FilterAndFindFMX</code>	Column filters, the filter row, the find panel, the column chooser and a saved layout on a large table.
<code>GridAndChart</code>	A chart bound to a grid on an <code>sgcDashboard</code> .
<code>GroupAndSummary</code> , <code>GroupAndSummaryFMX</code>	Grouping on two levels, group and footer summaries, frozen columns and live prices from a worker thread.
<code>LiveFeed</code>	A grid fed by a WebSocket stream through the <code>sgcWebSockets</code> bridge.
<code>LiveTicker</code> , <code>LiveTickerFMX</code>	A worker thread writing about a thousand price updates per second with the thread-safe <code>UpdateRow</code> .
<code>MigrateFromDBGrid</code>	Moving from <code>TDBGrid</code> and <code>cxGrid</code> to <code>sgcGrid</code> , concept by concept.
<code>ServerModeAndCards</code> , <code>ServerModeAndCardsFMX</code>	Server mode over a large table and the card view.
<code>Spreadsheet</code> , <code>SpreadsheetFMX</code>	Excel mode with sheets, formulas and per cell formats, saved to and loaded from XLSX.
<code>TreeAndDetail</code>	An org chart shown as a tree, and customers with a detail band of their orders.

Some of them use features that are not in the Community edition, such as editing, Excel export, tree and detail rows, server mode, the card view and the `sgcWebSockets` bridge. Each `readme.html` says what the demo needs.

Help topics

The sgcUI help is online at <https://www.esegece.com/help/sgcUI>. The sgcGrid topics are in its Grid section. Good places to continue:

Topic	What it covers
Architecture	The layers of sgcGrid and the threading model.
Editions	What each edition includes, feature by feature.
Platforms	Delphi versions, frameworks and packages.
Live Updates	Updating the grid from worker threads.
DataSet Binding	Everything about <code>TsgcGridSource_DataSet</code> .
Conditional Formatting	Format rules, operators, icons and icon sets.
Grouping	Group levels, the group panel and group summaries.
Migrating from TDBGrid	Importing the columns of a DB grid and the property map.
Troubleshooting	More problems and their fixes.

The Features folder of the Grid section has one topic per feature, from sorting and filtering to printing, undo and right-to-left layouts, and the Components folder has one topic per class.

On the website

- <https://www.esegece.com/products/sgcgrid/>: the product page, with the features, where sgcGrid is included, the demos and the trial download.
- <https://www.esegece.com/help/sgcUI>: the online help of the sgcUI suite.

Getting help

If you are stuck, write to info@esegece.com or open a ticket in the helpdesk of your account at <https://www.esegece.com/>. It helps to include your Delphi version, your edition, the demo or a small project that shows the problem, and the exact error message.

Enjoy building your first grid with sgcGrid.

© eSeGeCe. sgcGrid and sgcUI are products of eSeGeCe. <https://www.esegece.com>